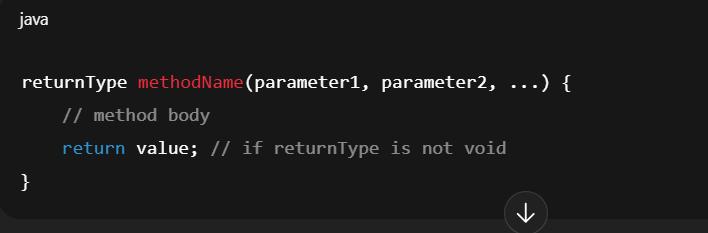
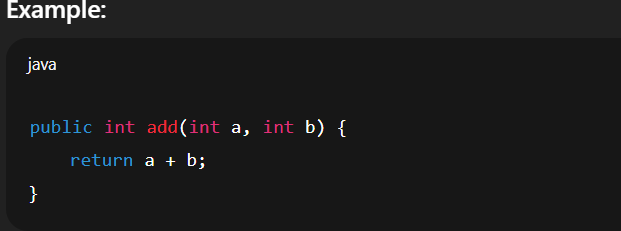
## Day 3-(Methods)

## ****What is a Method in Java?****

* A **method** is a block of code that performs a specific task.
* It helps in **code reusability**, **readability**, and **modularity**.
* You can call a method whenever you need it, instead of writing the same code again and again.

## ****Syntax of a Method:****





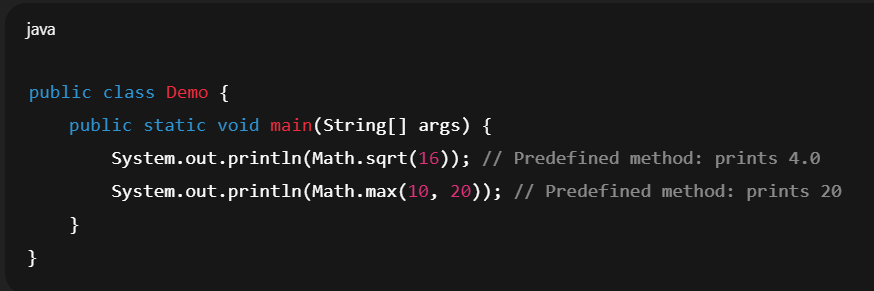
* public → access modifier
* int → return type
* add → method name
* (int a, int b) → parameters
* return a + b; → return statement

## ****Types of Methods in Java****

* **Predefined (Library) Methods**

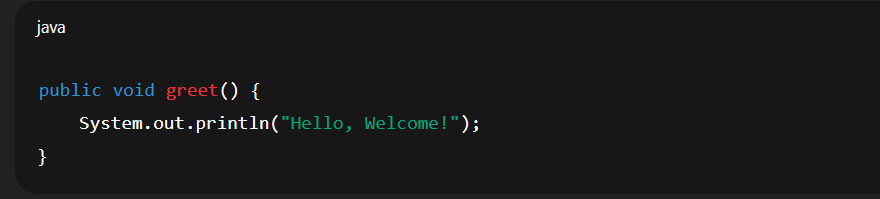
Already defined in Java libraries.

Example:



* **User-defined Methods**

Written by the programmer to perform specific tasks.

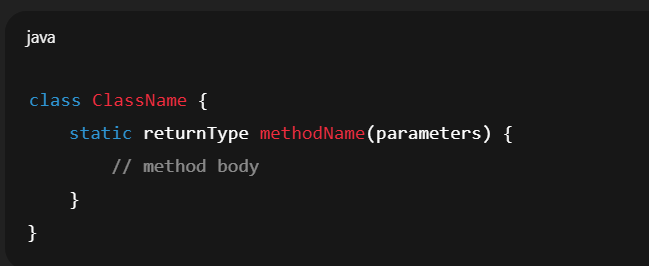


## ****Different Categories****

* **Static Method**

A **static method** belongs to the **class**, not to any specific object.  
That means you can call it **without creating an object** of the class.

* **Syntax**



### Key Points:

* Declared using the static keyword.
* Can be called directly using the **class name** (ClassName.methodName()).
* Cannot directly access **instance variables** or **instance methods** (because they belong to objects).



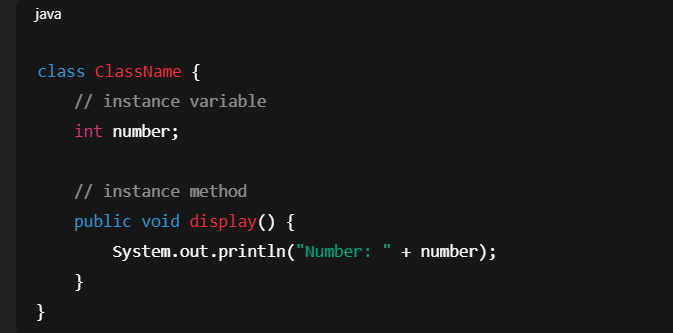
## 

## ****Instance Method****

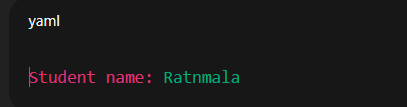
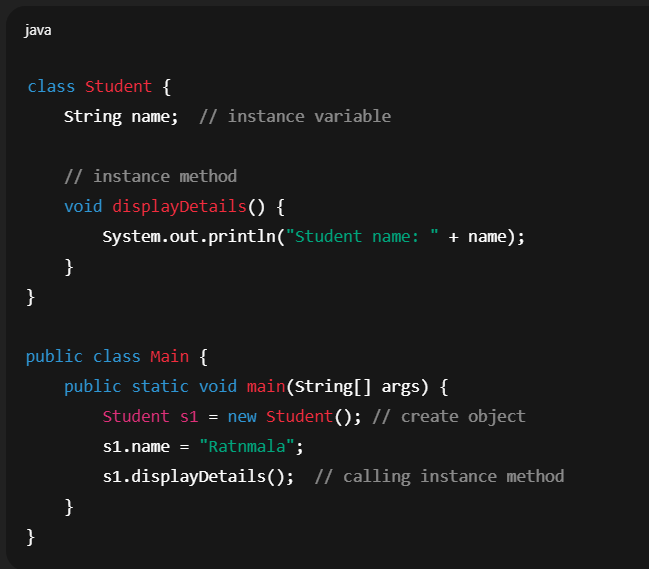
## An ****instance method**** is a ****non-static method**** that belongs to an object (instance) of a class.

## To call it, you need to ****create an object**** of that class.

* Instance methods can access:
* **Instance variables** (non-static fields)
* **Static variables** (class-level fields)
* **Syntax**



* Example



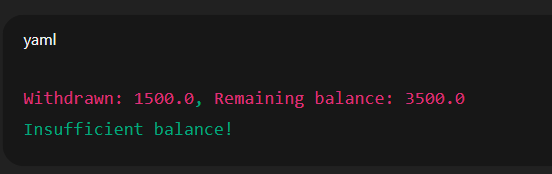
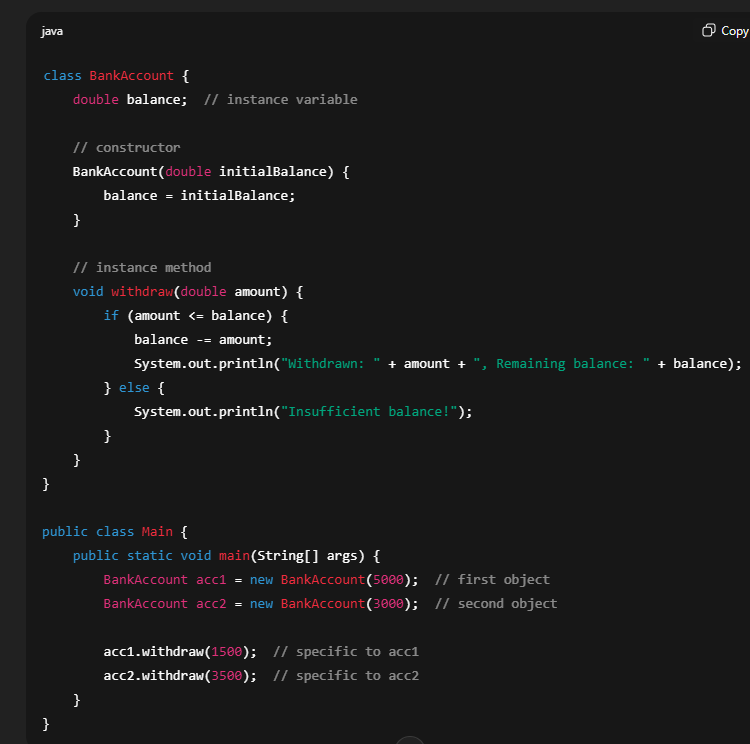
## ****Key Points****

* Declared **without the** static **keyword**.
* Must be called using **object reference**.
* Example: objectName.methodName().
* Can access both **instance variables** and **static variables**.
* Used when the behavior of a method depends on the **individual object’s state**.

## ****When to Use Instance Methods?****

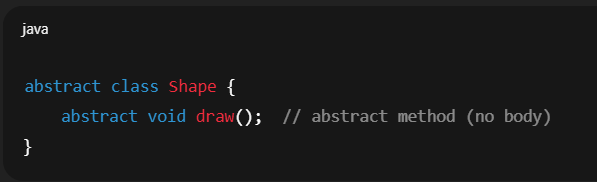
When each object of a class should have its own behavior.

* Example:
* A Car object → startEngine() depends on that specific car’s state.
* A BankAccount object → withdraw(amount) depends on that account’s balance.



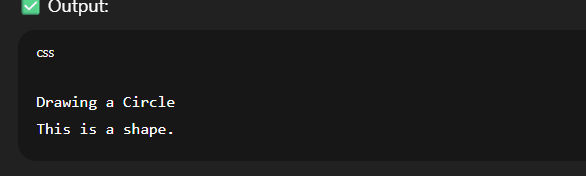
## ****Abstract Method****

* An **abstract method** is a method that is **declared without implementation** (no body).
* It is meant to be **overridden in subclasses**.
* Declared using the abstract keyword.
* Can only exist **inside an abstract class or interface**.
* Syntax



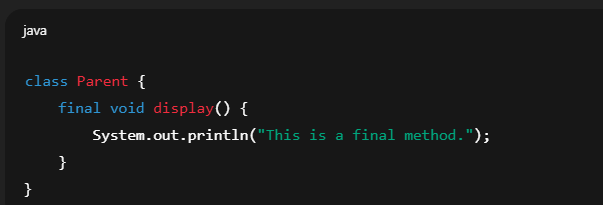
* Example





## ****Rules for Abstract Methods****

* Declared using the abstract keyword.
* No body → only method signature.
* Must be inside an **abstract class** or **interface**.
* Subclass must **override** the abstract method, unless the subclass is also abstract.
* Cannot be private, final, or static (because they need overriding).
* **Why Use Abstract Methods?**
* To enforce a **common contract** that subclasses must follow.
* To support **polymorphism** (different implementations of the same method).
* Example:
* Shape → draw() method (Circle draws a circle, Rectangle draws a rectangle).
* Payment → pay() method (CreditCard, UPI, PayPal each implement differently).
* Final Method
* A final **method** in Java is a method that **cannot be overridden** by subclasses.
* It is declared using the keyword **final**.
* You can still **inherit** it, but you **cannot change its implementation** in child classes.
* Syantax

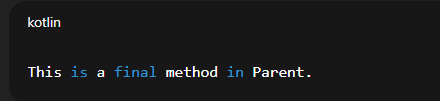


## Why use a final method?

* **To prevent overriding** when you want to keep the same logic for all subclasses.  
  (e.g., security-related methods, business rules, etc.)
* **For consistency** – ensures no child class changes its behavior.
* **Performance** – JVM can optimize **final** methods because it knows they won’t change.
* Example



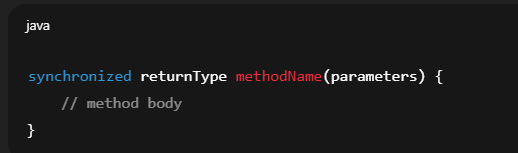
* Output

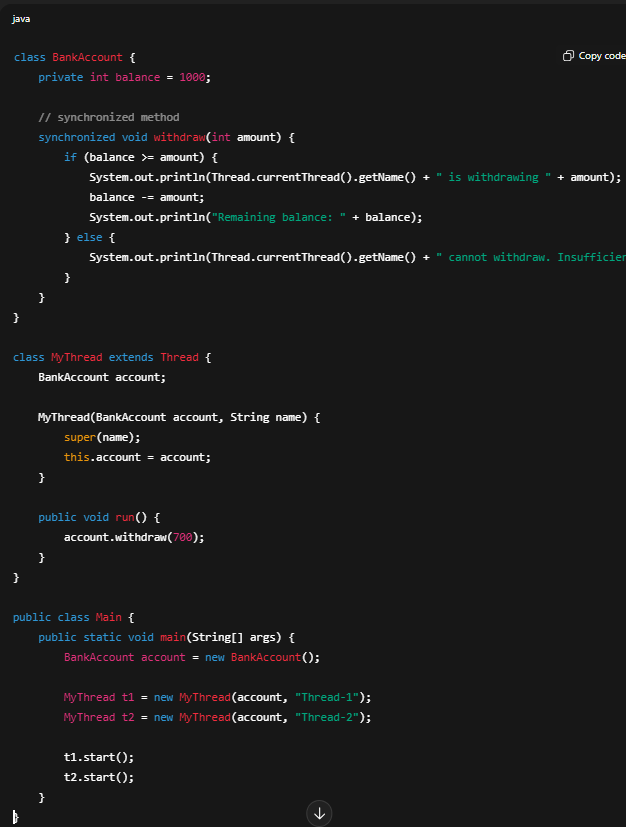


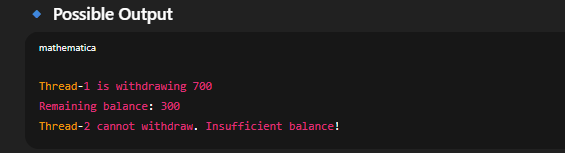
* If you try to **override** show() in Child, the compiler will give an **error**:  
  Cannot override the final method from Parent.

# synchronized Method

* A **synchronized method** in Java ensures that **only one thread at a time** can access that method of an object.  
  This is used in **multithreading** to prevent **data inconsistency** when multiple threads try to modify shared resources.
* Why use synchronized?
* If multiple threads access the same object at the same time, data can become corrupted.
* Using synchronized locks the method, so only **one thread enters**, others must **wait**.
* Syntax







* Here, because the method is **synchronized**, only one thread at a time can withdraw money.
* Without synchronized, both threads may withdraw at the same time, leading to **negative balance (data corruption)**.

